Module-1 | Batch-7

DevOps Workflow

What is DevOps?

DevOps is a collaborative approach to software development that emphasizes communication, integration, and cooperation between software developers and IT operations professionals. The goal of DevOps is to streamline the development, deployment, and operations of software by automating processes, improving team collaboration, and fostering a culture of continuous improvement.

Core Principles of DevOps

1. **Automation**: Automation plays a central role in DevOps, from code deployment to infrastructure management. By automating tasks such as code builds, testing, deployment, and infrastructure provisioning, teams can improve efficiency, reduce human error, and release software faster.
2. **Continuous Integration and Continuous Delivery (CI/CD)**:
    - **Continuous Integration (CI)** involves frequently integrating code changes into a shared repository, triggering automated builds and tests. This helps catch errors early and ensures code is always in a deployable state.
    - **Continuous Delivery (CD)** goes a step further by automating the deployment of code to production, ensuring software can be released at any time without manual intervention.
3. **Infrastructure as Code (IaC)**: DevOps teams manage infrastructure (servers, networks, databases) using code, allowing them to version control, automate, and replicate configurations across different environments. Tools like Terraform, Ansible, and AWS CloudFormation enable IaC.
4. **Monitoring and Feedback Loops**: DevOps emphasizes the use of real-time monitoring, logging, and analytics to gain insights into application performance and user behavior. This feedback is used to improve applications continuously and address issues proactively.
5. **Collaboration and Culture**: DevOps is as much about culture as it is about tools and processes. It encourages transparency, shared responsibility, and open communication, helping break down the traditional silos between development and operations teams.

Teams in DevOps

A successful DevOps ecosystem involves various teams working together, each with specific responsibilities and contributions to the software lifecycle. Here's a breakdown of the primary teams involved in DevOps:

1. Developers

Developers are primarily responsible for writing, testing, and maintaining the code that powers applications. They play a key role in DevOps, focusing on:

- **Coding**: Writing high-quality code in collaboration with DevOps teams to ensure it can be efficiently built, tested, and deployed.
- **Unit Testing**: Developers perform initial testing of their code to ensure that it functions as expected before integrating with other components.
- **Feature Development**: Developers work closely with product teams to develop and deliver features based on customer needs.
- **Collaboration on CI/CD**: Developers work with DevOps engineers to create CI/CD pipelines that automate testing and deployment, ensuring code reaches production efficiently.

Key Skills for Developers in a DevOps Environment:

- Familiarity with CI/CD tools (e.g., Jenkins, GitLab CI/CD)
- Knowledge of testing frameworks and practices (e.g., JUnit, pytest)
- Version control systems (e.g., Git)
- Understanding of containerization (e.g., Docker) and orchestration (e.g., Kubernetes)

2. Infrastructure Team (Ops Team)

The Infrastructure team, or IT operations team, manages the physical and virtual resources needed to support application deployment. They ensure the infrastructure is stable, secure, and scalable to meet application demands. Key responsibilities include:

- **Provisioning and Managing Infrastructure**: This involves setting up servers, storage, networking, and other resources that applications need.
- **Security and Compliance**: The infrastructure team ensures systems meet security standards, performing tasks such as patching, firewall configuration, and vulnerability management.
- **Monitoring and Troubleshooting**: They monitor infrastructure health, usage, and performance, ensuring systems remain operational and troubleshooting issues as they arise.
- **Infrastructure as Code (IaC)**: The team manages infrastructure configurations programmatically, using tools like Terraform and Ansible to ensure consistency across environments.

Key Skills for Infrastructure Teams in DevOps:

- Experience with IaC tools (e.g., Terraform, CloudFormation)
- Understanding of cloud services (e.g., AWS, Azure, GCP)
- Knowledge of security practices and compliance standards
- Expertise in monitoring and logging tools (e.g., Prometheus, Grafana)

3. DevOps Team

The DevOps team bridges the gap between development and operations, focusing on building and maintaining pipelines, automating workflows, and enhancing collaboration. This team usually consists of **DevOps Engineers**, **Developers**, and **Testers** specialized in DevOps practices. The responsibilities of the DevOps team include:

- **Pipeline Development and CI/CD Implementation**: DevOps engineers create and manage CI/CD pipelines that automate the build, test, and deployment process, enabling faster and more reliable software releases.
- **Automation and Scripting**: They write scripts to automate various processes, such as code deployment, infrastructure provisioning, and configuration management.
- **Monitoring and Incident Response**: DevOps engineers implement monitoring tools and systems to provide insights into application performance, helping detect and respond to incidents quickly.
- **Security Integration (DevSecOps)**: DevOps engineers work closely with security teams to incorporate security practices into CI/CD pipelines, ensuring security is considered at every stage of development.

Key Roles within the DevOps Team:

- **DevOps Developers**: These developers specialize in building infrastructure and automation tools. They work on developing scripts, managing IaC, and enhancing CI/CD processes.
- **DevOps Testers**: Testers in the DevOps team focus on automated testing, quality assurance, and test environment setup. They create automated tests to run in CI/CD pipelines, ensuring code quality and performance.

Key Skills for DevOps Engineers:

- Expertise in CI/CD tools (e.g., Jenkins, GitLab CI/CD)
- Proficiency in scripting languages (e.g., Python, Bash)
- Experience with cloud platforms and services (AWS, Azure, GCP)
- Familiarity with configuration management tools (e.g., Ansible, Chef)
- Knowledge of containerization and orchestration (Docker, Kubernetes)

Key Concepts in DevOps

1. Continuous Integration (CI)

Definition: Continuous Integration (CI) is the practice of integrating code into a shared repository frequently. Each code integration triggers an automated build and test sequence, ensuring the new code doesn't break existing functionality.

Example: A team uses GitLab CI/CD for Continuous Integration. Each time a developer pushes code to the repository, the CI system automatically compiles the code, runs unit tests, and alerts the developer of any issues before the code merges with the main branch. This process ensures that code is always in a deployable state.

---

2. Continuous Delivery (CD)

Definition: Continuous Delivery automates the release process, ensuring that code is always in a state where it can be deployed to production. This means that every code change that passes all stages of the CI pipeline is ready for deployment.

Example: A company uses AWS CodePipeline to automate deployment after successful integration. Once the code passes all tests, CodePipeline automatically pushes the code to a staging environment for final checks. With one click, the code can then be promoted to production, reducing manual deployment steps.

---

3. Continuous Deployment

Definition: Continuous Deployment goes one step further than Continuous Delivery by automatically deploying code to production as soon as it passes all required tests. This approach reduces human intervention and allows for more frequent updates.

Example: Facebook uses continuous deployment to push updates to its application frequently. When developers commit changes, they are tested and then deployed directly to production, allowing Facebook to release small, incremental changes regularly and maintain rapid feature delivery.

---

4. Infrastructure as Code (IaC)

Definition: IaC is the practice of managing and provisioning computing infrastructure through machine-readable code rather than through physical hardware configuration or interactive configuration tools.

Example: A team uses Terraform to define and provision cloud infrastructure on AWS. The infrastructure configurations, including servers, databases, and networks, are stored in version-controlled Terraform files, allowing them to recreate the entire environment from scratch in a new region if needed.

---

5. Configuration Management

Definition: Configuration management is the practice of handling changes systematically so that a system maintains integrity over time. It includes configuring servers, software, and environments in a consistent and repeatable manner.

Example: Using Ansible, a team manages configurations for a fleet of servers. With Ansible playbooks, they can apply the same configurations across multiple servers, ensuring consistency and making it easy to roll back changes if needed.

---

6. Monitoring and Logging

Definition: Monitoring and logging provide visibility into application and infrastructure performance, enabling teams to detect issues early, troubleshoot faster, and gain insights into user behavior and system health.

Example: Netflix uses tools like Prometheus for monitoring and ELK (Elasticsearch, Logstash, Kibana) for logging. These tools help track application performance, allowing Netflix engineers to proactively identify bottlenecks and reduce downtime, especially during peak times.

---

7. Collaboration and Communication

Definition: Collaboration and communication are vital in DevOps, breaking down silos between development and operations teams. DevOps encourages transparency, shared responsibilities, and alignment on goals.

Example: A team at Spotify uses Slack and shared dashboards in Grafana to keep everyone informed about deployment status, incidents, and key metrics. These tools foster open communication and allow everyone to contribute to issue resolution and process improvement.

Core Principles of DevOps

1. Automation

Description: Automation is at the core of DevOps. By automating repetitive tasks, teams can increase efficiency, reduce human error, and ensure consistency. Automation applies to testing, deployment, infrastructure provisioning, monitoring, and more.

Example: At Google, developers use automated pipelines for CI/CD to handle code builds, tests, and deployments. Tools like Jenkins and GitLab CI/CD allow teams to automate these processes, freeing developers to focus on writing quality code.

---

2. Continuous Integration and Continuous Delivery (CI/CD)

Description: CI/CD focuses on frequently integrating code and automating the delivery pipeline. It allows code to be tested and deployed in smaller, manageable increments, reducing risks and improving release quality.

Example: Amazon uses CI/CD to manage deployments for its global platform. With automated tests and builds, Amazon ensures that every code change is thoroughly vetted and can be deployed with minimal risk.

---

3. Infrastructure as Code (IaC)

Description: IaC enables teams to manage infrastructure using version-controlled code. This approach makes infrastructure deployments repeatable, consistent, and scalable, reducing the risks associated with manual configurations.

Example: Capital One uses IaC to deploy its infrastructure across AWS regions. By defining infrastructure in code (e.g., using AWS CloudFormation), the team can quickly replicate the same environment for testing, development, and production, ensuring consistency across deployments.

---

4. Microservices Architecture

Description: A microservices architecture breaks down applications into small, independent services that communicate over APIs. This modularity enables teams to work on individual components without impacting the entire application, allowing faster and more reliable deployments.

Example: Uber operates its platform using a microservices architecture. Each service, such as ride pricing, mapping, or payment processing, operates independently. If there's an issue with the payment service, other services remain unaffected, ensuring minimal disruption.

---

5. Configuration Management and Version Control

Description: Configuration management helps maintain system configurations across various environments. Version control, on the other hand, tracks changes in code, infrastructure configurations, and pipelines, enabling quick rollbacks and consistency.

Example: Git is a popular version control system used by DevOps teams to manage code and configurations. Using Git for IaC scripts allows teams to track every infrastructure change, facilitating rollback and collaboration.

---

6. Security and Compliance (DevSecOps)

Description: DevSecOps integrates security into every phase of the development and operations cycle. It encourages secure coding practices, vulnerability scanning, compliance checks, and access controls as part of the CI/CD pipeline.

Example: A financial institution uses DevSecOps practices to ensure secure software releases. Security scans, penetration testing, and compliance checks are automated in the CI/CD pipeline, ensuring the application meets regulatory standards before each release.

---

7. Monitoring, Logging, and Feedback Loops

Description: Monitoring and logging give visibility into application performance and issues. Feedback loops allow teams to analyze metrics and insights, using them to make improvements and optimize performance.

Example: Shopify uses monitoring tools like New Relic and Datadog to track application performance in real-time. Log data from these tools helps identify slowdowns or potential issues, allowing engineers to respond quickly and keep the application stable.

---

8. Collaboration and Communication

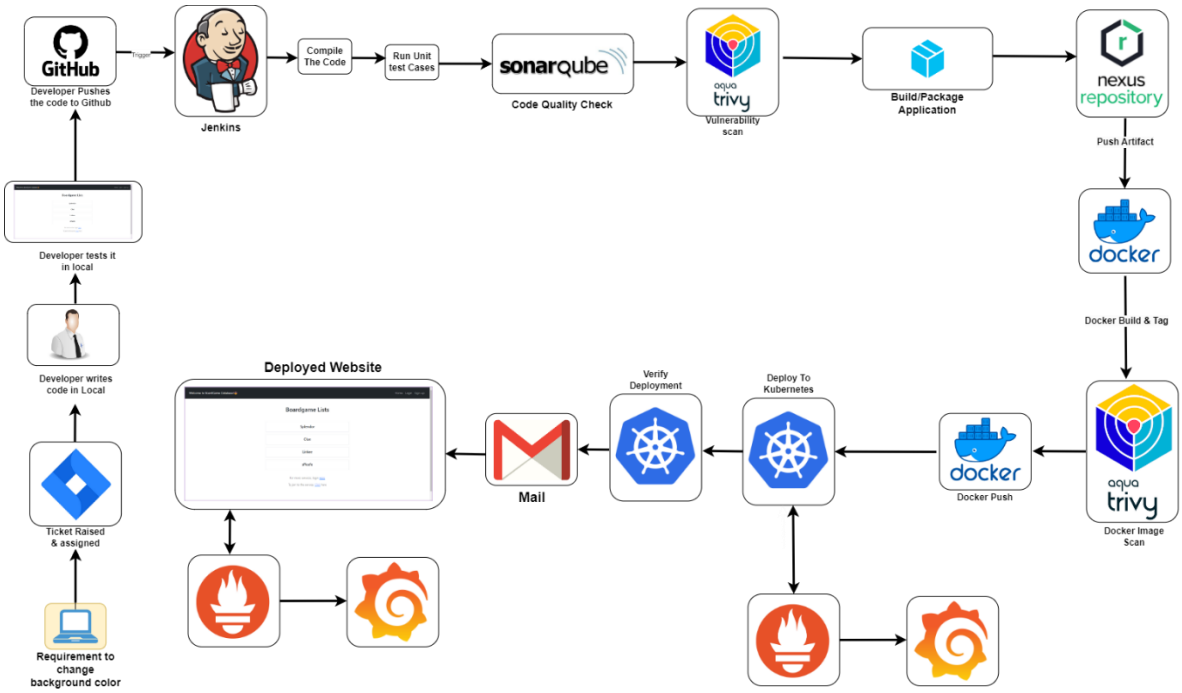
Description: Collaboration is a key principle of DevOps, fostering a culture where developers and operations teams share ownership of the entire lifecycle of applications. Tools that facilitate communication are essential for building and maintaining this culture.

Example: Atlassian promotes a collaborative culture by using tools like Jira for tracking work and Confluence for documenting processes. This allows everyone in the organization to stay informed and work together on shared goals.

Real-World Examples of DevOps in Action

1. **Netflix**:
    - Uses CI/CD pipelines for deploying code rapidly.
    - Leverages monitoring and feedback loops to improve user experience.
    - Implements microservices architecture, allowing the system to scale independently.
2. **Walmart**:
    - Uses IaC to manage infrastructure across its global data centers.
    - Automates deployments to handle large-scale traffic demands.
    - Uses monitoring tools to gain insights into system performance and proactively address issues.

DevOps Workflow



Declarative: Tool Install	Git Checkout	Compile	Test	File System Scan	SonarQube Analsyis	Quality Gate	Build	Publish To Nexus	Build & Tag Docker Image	Docker Image Scan	Push Docker Image	Deploy To Kubernetes	Verify the Deployment	Declarative: Post Actions
26s	4s	14s	21s	26s	18s	605ms	20s	22s	22s	28s	21s	1s	952ms	4s
26s	4s	14s	21s	26s	18s	605ms (paused for 5s)	20s	22s	22s	28s	21s	1s	952ms	4s

1. **Client Request for New Feature**
    - The client informs the project manager about a new feature that needs to be added to the application.
2. **Ticket Creation**
    - The project manager creates a ticket in a project management tool (e.g., Jira) to track the new feature request.
    - The ticket is then assigned to a developer for implementation.
3. **Code Development and Local Testing**
    - The developer writes the code for the new feature and tests it locally to ensure functionality and stability.
4. **Code Push to GitHub**
    - Once the developer is satisfied with the local testing, they push the code changes to a GitHub repository.
    - This triggers the CI/CD pipeline in Jenkins.

CI/CD Pipeline Stages in Jenkins

1. **Tool Installation**
    - Jenkins installs any necessary tools specified in the pipeline configuration. This could include tools for building, testing, and scanning code.
2. **Git Checkout**
    - Jenkins checks out the latest code from the GitHub repository, ensuring the pipeline works with the updated code.
3. **Code Compilation**
    - The pipeline compiles the code, converting it into an executable format, ready for further testing and packaging.
4. **Unit Testing**
    - Jenkins runs unit test cases to validate the individual components of the code. This ensures that any changes made have not broken existing functionality.
5. **SonarQube Analysis**
    - The pipeline runs a code quality analysis using SonarQube. This step checks the code for any bugs, vulnerabilities, and code smells, ensuring high code quality standards.
6. **File System Scan with Trivy**
    - Trivy performs a file system scan on the application code to identify any known vulnerabilities. This step enhances security by detecting potential threats early.
7. **Build/Package Application**
    - Jenkins builds or packages the application, preparing it for deployment. This could involve generating a JAR, WAR, or any other executable package format.
8. **Publish to Nexus Repository**
    - The built artifacts are published to a Nexus artifact repository. This makes them accessible for future deployment stages or other development environments.
9. **Docker Image Build**
    - Jenkins builds a Docker image for the application. This image encapsulates the application and its dependencies, making it portable and ready for containerized environments.
10. **Docker Image Scan with Trivy**

- Trivy performs a security scan on the Docker image to detect vulnerabilities in the image layers, ensuring a secure deployment.

11. **Push Docker Image to Registry**

- The Docker image is pushed to a Docker registry (e.g., Nexus or Docker Hub), making it accessible for deployment in containerized environments like Kubernetes.

12. **Update Kubernetes Manifests**

- Jenkins updates the Kubernetes manifest YAML files to reflect the new Docker image version or other configuration changes.

13. **Deployment to Kubernetes**

- The updated manifests are deployed to a Kubernetes cluster. This can involve creating or updating Kubernetes resources such as pods, services, and deployments.

14. **Verify Deployment**

- Jenkins verifies the deployment by checking the health of the deployed application in Kubernetes, ensuring it is running correctly and accessible.

15. **Email Notifications**

- Jenkins sends email notifications to relevant stakeholders (e.g., developers, project managers) regarding the status of the deployment. This keeps everyone informed of the deployment status.

Pipeline Summary (Execution Times)

The execution times for each stage in the pipeline are as follows:

- **Tool Install**: 26s
- **Git Checkout**: 4s
- **Compile**: 14s
- **Unit Test**: 21s
- **File System Scan**: 26s
- **SonarQube Analysis**: 18s
- **Quality Gate**: 605ms (paused for 1s)
- **Build**: 20s
- **Publish to Nexus**: 22s
- **Build & Tag Docker Image**: 22s
- **Docker Image Scan**: 28s
- **Push Docker Image**: 21s
- **Deploy to Kubernetes**: 1s

- **Verify Deployment**: 952ms
- **Post Actions**: 4s

Tools Used in Each Stage

- **Version Control**: GitHub for code storage and triggering CI/CD
- **CI/CD Automation**: Jenkins for orchestrating the pipeline stages
- **Static Code Analysis**: SonarQube for code quality checks
- **Artifact Repository**: Nexus for storing application artifacts
- **Containerization**: Docker for building and managing images
- **Security Scanning**: Trivy for file system and Docker image vulnerability scans
- **Container Orchestration**: Kubernetes for deploying and managing containerized applications
- **Monitoring**: Prometheus and Grafana for application health and performance monitoring
- **Notifications**: Email service for alerting stakeholders of deployment status

Deployment strategies

Deployment strategies in DevOps define how new versions of an application are released to users. Each strategy has its own advantages and drawbacks, and the choice of a strategy depends on factors such as the need for zero downtime, risk tolerance, rollback requirements, and complexity of the application.

1. Recreate (Single Deployment)

Description: In the recreate strategy, the existing version of the application is taken down completely, and the new version is deployed in its place. This approach causes some downtime as users cannot access the application during the deployment.

Best for: Small applications or non-critical applications where brief downtime is acceptable.

Example: A small e-commerce website can use the recreate strategy during low-traffic hours to deploy a new version. Since the downtime is short and scheduled, it minimizes disruption to users.

---

2. Rolling Deployment

Description: In a rolling deployment, instances of the old version are gradually replaced with instances of the new version. This strategy provides zero downtime and a continuous, seamless deployment experience.

How it Works:

- A few instances of the old version are terminated and replaced with instances of the new version.
- This process continues until all instances are running the new version.

Best for: Applications requiring zero downtime, such as web applications with multiple instances.

Example: An API-based application with several instances can be deployed with a rolling update. Kubernetes and other orchestration tools often provide rolling deployment as a standard strategy, ensuring no downtime while gradually replacing instances.

---

3. Blue-Green Deployment

Description: Blue-Green Deployment involves two identical environments: one running the old version (Blue) and another with the new version (Green). Traffic is initially directed to the Blue environment. Once the Green environment is ready and tested, traffic is switched from Blue to Green.

Benefits:

- **Zero downtime**: Users don't experience downtime because the switch between Blue and Green is instantaneous.
- **Easy rollback**: If issues are detected, it's easy to revert traffic back to the Blue environment.

Best for: Applications with stringent uptime requirements, large applications, or complex systems where rollback simplicity is important.

Example: An online banking application could use a blue-green strategy to release updates. During deployment, users access the Blue environment. Once the Green environment is tested and verified, traffic is switched to it. If any issues arise, the bank can easily switch traffic back to the Blue environment.

---

4. Canary Deployment

Description: In a canary deployment, a small subset of users is initially routed to the new version, while the rest continue to use the old version. If the new version performs well without issues, traffic is gradually increased until all users are using the new version.

Benefits:

- **Low-risk testing**: Allows teams to test the new version on a small scale.
- **Gradual rollout**: Issues can be identified and fixed with minimal impact on users.

Best for: High-traffic applications, where a gradual rollout is essential to catch issues early.

Example: A social media platform may use a canary deployment to introduce a new feature to 5% of its users. If the feature works as expected, the rollout can gradually increase to 25%, 50%, and eventually 100% of users. This ensures a smooth and risk-mitigated release process.

---

5. A/B Testing

Description: In A/B testing, two versions of the application are deployed simultaneously, but with different features or configurations. Traffic is divided between the two versions, and metrics are gathered to assess which version performs better.

Benefits:

- **Data-driven decisions**: Allows teams to gather real-world data on feature performance.
- **User feedback**: Direct feedback from users can influence which version to keep.

Best for: Applications with a strong focus on user experience, where testing different UI/UX elements or features can drive improvements.

Example: An e-commerce website might use A/B testing to compare two versions of its checkout process. Version A has a single-page checkout, while Version B has a multi-step checkout. Metrics like conversion rates are gathered, and the more successful version is kept in production.

---

6. Shadow Deployment

Description: In a shadow deployment, the new version runs alongside the old version, but only receives a copy of the live traffic without affecting end users. This allows the team to monitor how the new version handles the production load without impacting real users.

Benefits:

- **Testing at scale**: Allows for performance and functionality testing with live traffic data.
- **No risk to users**: Since users interact only with the old version, any issues with the new version won't impact them.

Best for: Applications where it's crucial to test performance under real traffic conditions, such as high-stakes financial or healthcare applications.

Example: A stock trading platform could use shadow deployment to test a new algorithm. The new algorithm receives live traffic data but doesn't interact with users directly. This allows the platform to assess its performance and accuracy under real conditions without risking user trust.

---

7. Feature Toggles (Feature Flags)

Description: Feature toggles allow developers to enable or disable specific features in an application without deploying new code. This approach enables incremental feature rollout and easy rollback by toggling the feature off if issues arise.

Benefits:

- **Granular control**: Features can be enabled or disabled for specific users or environments.
- **Quick rollback**: Any issues with the new feature can be mitigated by toggling it off.

Best for: Applications with frequent releases or where experimentation with new features is common.

Example: A streaming service like Netflix may use feature toggles to release a new recommendation algorithm to a subset of users. If the new algorithm causes issues, the toggle can be switched off instantly, reverting users to the old algorithm.

---

8. Progressive Delivery

Description: Progressive delivery is a modern approach that combines canary deployments and feature toggles to incrementally roll out features. It involves releasing features gradually, targeting specific user segments, and using automation to monitor and scale releases based on performance metrics.

Benefits:

- **Controlled rollout**: Features are rolled out slowly to specific users and scaled based on performance.
- **Enhanced monitoring**: Continuous monitoring ensures that potential issues are identified before full deployment.

Best for: Large-scale applications where control and flexibility in deployment are critical, and where user feedback is essential to scaling features.

Example: A gaming company might use progressive delivery to release a new in-game feature to players. The feature is initially available to a small group, with real-time monitoring in place. As positive feedback and stable performance are observed, the feature is rolled out to more players.

---

9. Immutable Deployment

Description: Immutable deployment involves deploying a new instance of the application without modifying the existing instance. Instead of updating or patching the current instance, a fresh instance with the new code is created, and the old instance is replaced.

Benefits:

- **Consistency and reliability**: Since each deployment is a fresh instance, there are fewer chances of leftover configurations or dependencies from previous versions.
- **Easy rollback**: If issues arise, the old instance can be brought back quickly.

Best for: Stateless applications or containerized environments where clean, consistent deployments are critical.

Example: An organization using AWS EC2 for hosting applications might implement an immutable deployment strategy. For each new release, they deploy a new EC2 instance with the updated code and then switch traffic to the new instance. This ensures that each deployment is consistent and clean.

Comparison Table of Deployment Strategies

Strategy	Zero Downtime	Easy Rollback	Suitable For	Complexity
Recreate	No	Moderate	Small applications or non-critical apps	Low
Rolling	Yes	Harder	Multi-instance applications	Moderate
Blue-Green	Yes	Easy	Critical applications with high uptime	High
Canary	Yes	Moderate	High-traffic applications	High
A/B Testing	Yes	Moderate	User experience-focused applications	High
Shadow	Yes	Moderate	Applications needing testing with real traffic	High
Feature Toggles	Yes	Very Easy	Frequent releases or feature experimentation	Moderate
Progressive Delivery	Yes	Very Easy	Large applications needing controlled rollout	High

Strategy	Zero Downtime	Easy Rollback	Suitable For	Complexity
Immutable	Yes	Easy	Stateless applications	Moderate

Each deployment strategy has its unique advantages and best-use cases, depending on the application requirements, user base, and risk tolerance. By choosing the appropriate deployment strategy, teams can release updates efficiently, reduce downtime, and maintain high-quality user experiences.

Environments in DevOps

In DevOps, environments represent distinct stages where code is developed, tested, and deployed. Each environment serves a specific purpose in the software development lifecycle, ensuring that code is stable, secure, and performs as expected before reaching end-users.

1. Development (DEV) Environment

- **Purpose**: The DEV environment is primarily used for writing, debugging, and initial testing of code. It's where active development and experimentation happen, allowing developers to create and refine new features or fix bugs.
- **Characteristics**:
    - Frequent code changes and updates are made as developers work on new features.
    - Access is generally open to developers, with fewer restrictions on modifying the code.
    - Basic testing, often unit tests and code reviews, is conducted to check functionality and fix obvious issues.
- **Promotion Criteria**: Code is promoted from DEV to QA when initial tests (like unit tests) pass, and the code is deemed stable enough for more rigorous testing in a controlled environment.

---

2. Quality Assurance (QA) Environment

- **Purpose**: The QA environment is dedicated to comprehensive testing. QA engineers perform various tests to validate that the application meets functional, integration, and performance requirements.
- **Characteristics**:
    - More stable than DEV, with fewer code changes, making it easier to identify and reproduce issues.
    - Extensive automated and manual tests, such as functional, integration, and performance tests, are conducted here.
    - Configuration closely matches the production environment to simulate real-world conditions, allowing for accurate testing.
- **Promotion Criteria**: Code is promoted from QA to Pre-Production (PPD) when it has passed all QA tests and meets the predefined acceptance criteria set by the team.

---

3. Pre-Production (PPD) Environment

- **Purpose**: The PPD environment, also known as Staging, is a near-identical replica of the production environment. It is used for final testing and validation before the code goes live to ensure everything works seamlessly in a production-like setting.

- **Characteristics**:
    - Mimics the production environment closely, including configuration, data, and load, for accurate final testing.
    - User Acceptance Testing (UAT) is conducted by stakeholders to ensure the application meets their requirements.
    - Limited access is maintained to ensure stability, and only essential team members can make changes.
- **Promotion Criteria**: Code is promoted from PPD to Production (PROD) after it passes all final tests, including UAT and performance testing. Stakeholders must approve the release.

---

4. Production (PROD) Environment

- **Purpose**: The PROD environment is where the application is live and accessible to end-users. It's the final environment, where the software operates under real-world conditions, and any issues here directly impact the user experience.
- **Characteristics**:
    - The environment is highly stable and secure, with stringent access control to prevent unauthorized changes.
    - Monitored closely for performance, availability, and security to ensure optimal operation.
    - All changes are strictly controlled, reviewed, and approved before deployment to prevent service disruption.
- **Promotion Criteria**: Code is deployed to PROD after passing rigorous testing and receiving final approval from relevant stakeholders, including product owners and possibly external clients.

---

5. Disaster Recovery (DR) Environment

- **Purpose**: The DR environment is a backup environment designed to ensure business continuity in case of catastrophic failures in the PROD environment. DR enables quick recovery and minimizes data loss during unforeseen events.
- **Characteristics**:
    - Kept in sync with the PROD environment, often in a separate geographical location to ensure resiliency.
    - Configurations, data, and infrastructure mirror PROD, allowing it to take over seamlessly if PROD goes down.
    - Regular DR tests and drills are conducted to ensure readiness and verify that the DR environment can handle the load in case of failover.

- **Promotion Criteria**: The DR environment is not typically part of the regular promotion process. Instead, it is updated continuously to match the PROD environment. It is activated only during an actual disaster scenario.

Promotion Process Between Environments

The promotion process defines how code moves from one environment to the next, ensuring quality and stability at each step.

1. From DEV to QA

- **Process**:
    - Developers commit code to the version control system (e.g., Git).
    - Automated Continuous Integration (CI) processes are triggered to build the code and run initial tests.
    - Once initial tests pass, the code is automatically deployed to the QA environment.
- **Purpose**: This step ensures that only stable, tested code moves from DEV to QA, preventing bugs from affecting later stages.

---

2. From QA to Pre-Production (PPD)

- **Process**:
    - QA engineers conduct extensive testing, including automated and manual tests, covering functionality, integration, and performance.
    - Any identified issues or bugs are fixed by developers, and the code is retested to confirm stability.
    - Once the code passes all tests and meets acceptance criteria, it is reviewed and approved for deployment to the PPD environment.
- **Purpose**: This step confirms that the code is stable, meets requirements, and is ready for final validation in a production-like environment.

---

3. From Pre-Production (PPD) to Production (PROD)

- **Process**:
    - User Acceptance Testing (UAT) is performed in the PPD environment by stakeholders to validate the functionality and user experience.
    - Final performance and load tests are conducted to ensure the application can handle expected real-world usage.
    - Once all tests pass and stakeholders approve the release, the code is deployed to the PROD environment.

- **Purpose**: This final validation ensures that the release meets user expectations and is fully prepared for real-world use.

---

4. Disaster Recovery (DR) Updates

- **Process**:
    - Scheduled syncs or replication processes keep the DR environment up to date with the latest changes from the PROD environment.
    - Regular DR drills and tests are conducted to ensure that the DR environment can take over quickly in the event of a production failure.
    - Updates to the DR environment include database synchronization, configuration replication, and infrastructure readiness checks.
- **Purpose**: This ensures that the DR environment remains prepared to take over in an emergency, minimizing downtime and data loss.

Summary Table of Environments and Promotion Process

Environment	Purpose	Characteristics	Promotion Criteria
DEV	Active development and experimentation	Frequent code changes, basic testing, open access to developers	Code is stable enough for QA testing
QA	Comprehensive testing and validation	Stable, automated/manual tests, simulates PROD configuration	Code passes all tests and meets acceptance criteria
PPD	Final testing and validation before PROD	Mimics PROD, UAT and performance testing, limited access	Passes UAT, load tests, and stakeholder approval
PROD	Live environment for end-users	Highly stable, monitored for performance, changes are strictly controlled	Approval from stakeholders after all tests
DR	Backup for PROD in case of disaster	Kept in sync with PROD, includes configurations and data for quick restoration	Not part of promotion; activated during disaster

---

This promotion process and environment setup ensure that code is thoroughly tested, validated, and ready to meet user expectations before reaching the production environment. Additionally, the DR environment provides a fallback to minimize the impact of potential disasters, ensuring that the system remains resilient and reliable.